Attorney's Docket No.: 10559/387001 / P10192

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: RECOGNIZING SIGNALS IN DESIGN

APPLICANT: ARVIND B. IVER, DAVID J. HARRIMAN AND ARTHUR D. HUNTER

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EE647282668US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

December 27, 2000
Date of Deposit

Signature

Derek W. Norwood
Typed or Printed Name of Person Signing Certificate

RECOGNIZING SIGNALS IN DESIGN SIMULATION

BACKGROUND

The invention relates to recognizing signals in design simulation.

To determine whether a hardware or a system design will perform as intended, the design is often simulated to produce simulated signals. Specific sets of the simulated signals make up transactions or protocols. A transaction is a set of signals that indicate a unit of interactions such as a read or write operation. A protocol refers to signals that define rules of data transmission.

Simulated transactions or protocols conventionally are verified either by producing specific signal events and data during the simulation, which is time-consuming, or using an unstructured ad-hoc post-processing program on the simulation output. Simulations of lower-level designs are not easily extendible to higher-level designs, such as full-chip or system-level designs.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a flow of tracking and checking transactions.

FIG. 2 shows a table of rules and actions.

FIG. 3 shows a table of a partial signal event list.

FIG. 4 shows a time-evolving stack.

FIG. 5 illustrates how the stack is manipulated.

FIG. 6 illustrates a design simulation.

DETAILED DESCRIPTION

As shown in FIG. 1, a simulator 10 produces an event list file 20 which is a list of time-stamped signal events at the boundary of a design under test. The event list file 20 can be generated for every test by the simulator 10. The file 20 can be stored as character strings that are read from left to right.

A front-end reader called a unit tracker and checker (UTC) 30 reads the signal events and keeps track of the state of the events at each rising edge of a clock signal. Because signals are relevant at the clock rising edge for many logic circuit designs, the UTC 30 is configured to filter out signal events between clock cycles and to present the state of signal events at each clock rising edge as signal values 40. For example, assuming that the clock cycle is 10 nanoseconds (ns), if the signal event list shows that A = 1 and B = 1 at 121 ns and A = 0 at 123 ns, the UTC 30 presents A=0 and B=1 at the clock rising edge at 130 ns.

The signal values 40 are read by a recognition program 45 generated by a unit tracker and checker yet another compiler compiler (UTCYACC) program 50. The UTCYACC program incorporates production rules 60, each of which is coupled to a corresponding action 70, into the program 45. The program 45 uses the production rules and the actions 70 to recognize transactions or protocols in the signal values 40.

The production rules 60 are formal languages comprising atomic rules. An atomic rule interprets a set of atomic signal events, each of which is an indivisible unit of interaction, for example, between devices. Atomic signal events are found in the event list file 20. The production rules 60 include "read transaction" and "write transaction."

Additionally, each production rule 60 is accompanied by an action 70, which is a set of tasks to be executed whenever the production rule applies. One such action 70 involves printing out a transaction output 90 which may contain such information as event time, cycle identification, transaction and event identification. The action 70 can further print out details of a transaction, such as operation code (opcode), length, address, byte enable, and data, and may indicate whether the transaction was completed successfully or whether errors were incurred. Additionally, the action 70 can perform coherency checks and compute user-defined values for later use.

At a clock rising edge, the action 70 manipulates the atomic rules extracted from the signal values 40 using a stack 80. Details of how the stack 80 are used to recognize a transaction are described below.

Context-independent formal languages are used to model a transaction. The formal languages are a set of rules that define the transaction successively until atomic signal events are obtained. For example, FIG. 6 shows a unit (U) subject to testing. One of the transactions that the unit (U) can perform is a read operation from an output/input device controller (D) to a memory controller (M). In the following discussion, such a read transaction is designated symbolically as DMRead.

The device controller (D) presents a read command (DUcmd) to the unit (U). After several clock cycles, the unit (U) presents a command (UMcmd) to the memory controller (M). The memory controller (M) responds to the unit (U) with data (MUdata) after several more clock cycles. Then, the unit (U) responds to the device controller (D) with the data (UDdata).

The read transaction rule can be written as follows:

DMRead := DUcmd UMcmd MUdata UDdata.

The above rule includes a rule symbol (left-hand side) and production symbols (right-hand side). The rule is an example of a non-atomic rule that requires further definition. The device-to-unit read command, DUcmd, can be defined, for example, in terms of handshake signals, that is, atomic signal events, designated DUcavail and UDcget:

DUcmd := (DUcavail=1) AND (UDcget=1).

As shown in FIG. 6, the handshake signal DUcavail is transmitted from the device controller (D) to the unit (U). The unit (U) responds by sending the handshake signal UDcget to the device controller (D). The device-to-unit command, DUcmd, including signals such as device address and opcode, is then sent to the unit (U). In other words, if the handshaking signals are detected, the command signals (DUcmd) are sent from the device controller (D) to the unit (U). No further breakdown of this rule is required because atomic signal events have been obtained. Therefore, the device-to-unit command, DUcmd, is an example of an atomic rule.

Similarly, the unit-to-memory command, UMcmd, can be defined as follows:

UMcmd := (UMcavail=) AND (MUcget=1).

For the purpose of illustration, it is assumed that memory-to-unit data, MUdata, is transferred in two clock cycles and unit-to-device data, UDdata, is transferred in one to four clock cycles, depending on the data length. The

memory-to-unit data, MUdata, and the unit-to-device data, UDdata, can be defined as follows:

MUdata := MUdxfer MUdxfer,
UDdata := UDdxfer, and
UDdata := UDdata UDdxfer.

The MUdata rule specifies two data transfers. The two UDdata rules constitute a recursive definition and can specify any number of data transfers. The first UDdata is defined as one data transfer, and the second UDdata is defined as UDdata followed by a data transfer.

The non-atomic rules MUdxfer and UDdxfer can be defined as atomic rules using handshake signals as follows:

MUdxfer := (MUdavail=1) AND (UMdget=1), and
UDdxfer := (UDdavail=1) AND (DUdget=1).

The foregoing rules, along with corresponding actions, are summarized in FIG. 2.

The UTCYACC program 50 generates the recognition program 45, which applies the read transaction rules to the event list 20 generated by the simulator 10 to recognize the read transaction. The recognition program 45 is a stack-based engine that executes the user-defined action 70 that manipulates the stack 80 whenever the rule applies.

The manipulation of the stack 80 is shown in FIG. 5. The program 45 checks (100) whether each atomic rule applies to the signal events. If a particular rule applies, the program 45 places (110) the applicable rule symbol on top of the stack. A stack is a data structure for storing items that are to be accessed in last-in, first-out order.

The program 45 then reduces (120) the stack by applying the appropriate non-atomic rules. This involves comparing symbols on top of the stack to production symbols of the non-atomic rules. If the symbols on the stack match the production symbols (e.g. MUdxfer MUdxfer), the program 45 removes (130) the appropriate number of production symbols from the stack and pushes (140) the appropriate rule symbol (MUdata) onto the stack in their place.

When the stack contains a top-level symbol (e.g. DMRead), the transaction is recognized (150) and the stack is emptied.

If the end of the event list 20 is reached and the stack is not empty, the program 45 reports (160) that an error occurred and that the transaction was not completed.

FIG. 3 shows an example of a partial event list for a read transaction. The event list as shown is not filtered by the UTC 30. A clock period of 10 ns interval is assumed. FIG. 4 shows how the contents of the stack are modified according to the application of rules by the program 45 through this event list.

At time 40 ns, the program 45 recognizes the atomic events DUcavail=1 and UDcget=1 corresponding to DUcmd, and the command gets placed on top of the stack as shown in FIG. 4. At time 60 ns, another command (UMcmd) is recognized and is placed on top of the stack. At time 90 ns, a data transfer (MUdxfer) is recognized and is placed on the stack. At time 100 ns, a clock cycle later, another data transfer is recognized and is placed on top of the stack. The program 45 applies a non-atomic rule (see FIG. 2), removes MUdxfer MUdxfer from the stack and places MUdata on top of the stack.

At time 120 ns, a data transfer (UDdxfer) is recognized and placed on the stack. The non-atomic, first UDdata rule (see FIG. 2) is used to replace UDdxfer with UDdata. At time

130 ns, another UDdxfer is recognnized and is placed on the stack. The second UDdata rule (see FIG. 2) is applied to remove UDdxfer UDdata and to place UDdata on the stack. The program 45 then recognizes that the stacked symbols, UDdata MUdata UMcmd DUcmd, correspond to the transaction symbol DMRead and removes the symbols to place DMRead on top of the stack. Subsequently, DMRead is recognized as a top-level symbol, and the stack is emptied. The functions of removing symbols from the stack and placing them on the stack are executed by the actions 70.

In addition to manipulating the stack 80, the actions 70 specify user-defined tasks. This includes printing useful signal values, performing coherency checks, and computing other values of interest. For example, the action, DUcmd_act() can be coded to check if the transaction is a read command transfer, to obtain current stack/cycle identification for the transaction, to place DUcmd on the stack, to print cycle identification and signal values such as address, opcode and length, to compute the number of data transfers, and to increment the cycle identification for the next DUcmd event.

Similary, the action for UDdxfer can be coded to obtain current stack/cycle identification for the transaction, to count down the number of data transfers, to place UDdxfer on the stack, to print cycle identification and signal values, to check the data against what was obtained on the memory-to-unit data event, and to move to the next cycle if all data transfers are done.

The foregoing examples represent actions for atomic rules. Actions for a non-atomic rule, such as UDdata_act(n), can be coded to obtain current stack/cycle identification for the transaction, to remove n symbols from the stack and to place UDdata on the stack. Similary, the action for the non-

atomic rule DMRead_act() can be coded to obtain current stack/cycle identification for the transaction, to check to see that all unit-to-device data transfers are done, to remove four symbols from the stack and to print "Done" if the stack is empty.

Any non-empty stack is reported as an incomplete transaction and the contents of the stack are printed to indicate the events completed at that time.

The ordering of atomic rules can be important in some instances. For example, a DUcmd event occurs before a UMcmd event, and the rules should reflect this order. Similarly, non-atomic rules should be ordered to reduce the stack. For example, a DMRead rule should be applied after a DUdata rule is applied.

Some tests may have more than one instance of a transaction cycle. Each transaction can be tracked by using a separate stack that is uniquely identified by the cycle-identification. The cycle-identification is incremented when a new transaction begins. For example, two DUcmd events would imply that the second DUcmd should be assigned to the next stack.

If the design unit (U) also performs another transaction such as a write transaction, there should be separate rules and actions table for DMWrite. The UTCYACC program 50 can incorporte write transaction rules into the recognition program 45. Thus, the system would be reporting transaction events that they individually recognize.

If different transactions such as read or write transactions share the same signal events, such as DUcmd, then an opcode can be used to uniquely identify which transaction applies. An additional check on the opcode in the DUcmd rule can be implemented to address this type of situation.

To illustrate, it is assumed that there are two instances of read transaction with the event list below:

$DUcmd_1$ $DUcmd_2$ $UMcmd_2$ $UMcmd_1$...

The $UMcmd_2$ event for the second cycle occurs before the first and the recognition program 45 will incorrectly associate the $UMcmd_2$ event with the first stack. To address this situation, either an ordering algorithm can be implemented or the address bits can be used to identify the cycle uniquely. Alternatively, the internal signals in the event list in addition to those at the boundary may be used in the rules or actions to distinguish the events.

The unit (U) may retry an event based on a preferred condition, such as a time-out. In that case, the retry condition should be part of the rules 60. Alternatively, the program 45, internal signals, or unique signal values can be used to detect retry events.

A transaction rule may be represented by different permutations of production symbols. For example, assume that a rule for DMWrite can be expressed as follows:

DMWrite := DUcmd DUdata UMcmd UMdata.

It may be valid for the unit-to-memory command transfer to occur before any device-to-unit data transfer. Thus, the following rule is valid also:

DMWrite := DUcmd UMcmd DUdata UMdata.

As long as all permutations of the rule are specified, the stack-based approach is able to handle the different situations.

A set of rules and actions to detect an incorrect signal assertion or de-assertion can be created. Garbage collection rules can be used to detect invalid opcodes, invalid signal values during reset and hanging signals. These rules can be useful in increasing the coverage of the checks and can be extended to include all signals.

The foregoing techniques can be implemented in a program executable on a computer system. The program can be stored on a storage medium readable by a general or special purpose programmable computer system. The storage medium is read by the computer system to perform the functions described above.

The invention can also be implemented using digital logic hardware.

Other implementations are within the scope of the following claims.